https://www.infoq.com/articles/why-dsl-collection-anecdotes/

# Why DSLs? A Collection of Anecdotes

Key Takeaways

* Domain-specific languages allow non-developer domain experts to contribute directly to the software development process with "executable specifications" that are much more useful than prose user stories or requirements documents.
* From a software architect's perspective, DSLs allow the complete separation of the business logic from the implementation technology, thereby avoiding the trap of legacy systems.
* In contrast to popular belief, the approach works in practice, as this article shows through a collection of real-world stories that illustrate how companies benefit from the approach.
* DSLs can be used in systems engineering, healthcare as well as more classical business domains such as finance, insurance and payroll. Each language is different, but the idea, the infrastructure and the language tooling are the same.
* The automation of software development through code generation is an important benefit of DSLs. But there are others, including analyses, optimization, and, importantly, the much more meaningful inclusion of non-developer stakeholders into the process of software creation.

**Introduction**

*The future is already here --*  
*It's just not evenly distributed.*  
William Gibson

Or rather, one particular future is already here. The one in which domain-specific languages (DSLs) deliver significant improvements in productivity, software quality or architectural flexibility. But, paraphrasing Gibson, the approach is not used widely, so it's perceived as being something for the future. Or maybe as something that does not work at all.

In this article I discuss a number of example DSLs that are used successfully in real-world applications. I have personally been involved in the development of many of them, the others have been developed by people I know and trust. In this article I use short essays to discuss the languages. The essays are fictional in their composition and some details, but are faithful to what happened in the real projects. For several of the languages, more extensive and more systematic papers have been published, and I refer the reader to those papers at the end of each anecdote.

**RELATED SPONSORED CONTENT**

* [Secure Your Applications with Authentication and Authorization](https://www.infoq.com/vendorcontent/show.action?vcr=9fb33587-5111-4f0d-b9c9-274f96f3722d&primaryTopicId=2497&vcrPlace=EMBEDDED&pageType=ARTICLE_PAGE&vcrReferrer=https%3A%2F%2Fwww.infoq.com%2Farticles%2Fwhy-dsl-collection-anecdotes%2F&itm_source=infoq&itm_medium=VCRPopular&itm_campaign=vcr_articles_click&itm_content=embedded)
* [Getting Started with Progressive Web Apps (PWAs)](https://www.infoq.com/vendorcontent/show.action?vcr=5574&primaryTopicId=2497&vcrPlace=EMBEDDED&pageType=ARTICLE_PAGE&vcrReferrer=https%3A%2F%2Fwww.infoq.com%2Farticles%2Fwhy-dsl-collection-anecdotes%2F&itm_source=infoq&itm_medium=VCRPopular&itm_campaign=vcr_articles_click&itm_content=embedded)
* [Comprehensive Kubernetes Observability at Scale](https://www.infoq.com/vendorcontent/show.action?vcr=5588&primaryTopicId=2497&vcrPlace=EMBEDDED&pageType=ARTICLE_PAGE&vcrReferrer=https%3A%2F%2Fwww.infoq.com%2Farticles%2Fwhy-dsl-collection-anecdotes%2F&itm_source=infoq&itm_medium=VCRPopular&itm_campaign=vcr_articles_click&itm_content=embedded)
* [Best Practices for Secure Infrastructure Access (On-demand webinar) – Watch Now](https://www.infoq.com/vendorcontent/show.action?vcr=5551&primaryTopicId=2497&vcrPlace=EMBEDDED&pageType=ARTICLE_PAGE&vcrReferrer=https%3A%2F%2Fwww.infoq.com%2Farticles%2Fwhy-dsl-collection-anecdotes%2F&itm_source=infoq&itm_medium=VCRPopular&itm_campaign=vcr_articles_click&itm_content=embedded)
* [Detailed Review of JDK Implementations](https://www.infoq.com/infoq/url.action?i=131f2108-c01d-4c8e-845e-bf17a7957ebf&t=f)

**RELATED SPONSOR**

[](https://www.infoq.com/infoq/url.action?i=786739ce-1d5f-4eda-b35a-42a3cc265385&t=f)

**Modern App Development.**  
Supported enterprise software for Jakarta EE and MicroProfile applications. [**Learn more**](https://www.infoq.com/infoq/url.action?i=67f63924-bec9-4dcc-b468-b78a26202021&t=f).

This paper is the definition of "anecdotal evidence". I'm experimenting with this style in order to try to lower the bar as much as possible to get an impression why people use DSLs and how they are successful. Obviously, this paper is not the most scientific treatise on DSLs ever written. If you want to read a more technical discussion of many of these same topics I suggest you read the paper Fusing modeling and programming into language-oriented programming on the relationship between modeling and programming[1](https://www.infoq.com/articles/why-dsl-collection-anecdotes/#1), [The design, evolution, and use of kernelf](https://www.infoq.com/articles/why-dsl-collection-anecdotes/#13) on the design and use of the functional language, and [Lessons learned from developing mbeddr: a case study in language engineering with MPS](https://www.infoq.com/articles/why-dsl-collection-anecdotes/#11) on the experiences of building DSLs with MPS.

I will start with a brief intro to DSLs; if you have experience with DSLs I suggest you skip the next [section](https://www.infoq.com/articles/why-dsl-collection-anecdotes/#salaries).

**Domain-Specific Languages**

Because software plays a central role in an increasingly diverse array of fields, it is increasingly common that those people who know what the software should do are not the same as those who program the software. So for the software to work right, the domain expert's "brain contents" have to be formalized into an executable recipe we call a program. The traditional approach is that the domain expert communicates their requirements to the programmer in a more or less unstructured way: in conversations, through users stories, as Word documents, Excel files, or Doors databases. The developer reads this text, tries to understand it and then implement it correctly as a program. This process of formalization into code helps detect some ambiguities, inconsistencies or incompleteness; but ultimately, it is when the domain expert then plays with the finished software - or ideally writes [Gherkin-style acceptance tests](https://cucumber.io/docs/gherkin/) - that final validation happens.

**We all know that this approach doesn't work so well.**

Domain-specific languages rely on a different approach. They allow the domain expert to specify the behavior of the software directly. The transformation from *unstructured thought* to *executable specification* happens in their brains. The executable specifications - or models - created this way are then automatically transformed into "real" source code by machinery developed by software engineers.

Does this really work? It does under certain conditions. In particular, the language must be suitable for use by non-programmers. The primitives in the language should not be generic to "computation" - such as variables, conditions, loops, functions, monads or classes - but instead be specific to the domain, and therefore meaningful to the user: decision table, treatment step, tax rule or satellite telemetry message definition. The syntax should build on existing notations and conventions used in the domain - tables, symbols, diagrams and text - and not just consist of magenta-colored keywords and curly braces. DSLs are also usually less flexible in the sense that users can only compose new abstractions in very limited ways; while this would be a problem for general-purpose languages, it is a plus for DSLs because it ensures that programs are less complicated and easier for tools to analyze and provide IDE support for. This paper uses anecdotes to illustrate and explain the above claims. If you are interested in scientific studies, this is not the paper for you; instead this survey by Mernik et al. ([When and how to develop domain-specific languages](https://www.infoq.com/articles/why-dsl-collection-anecdotes/#3)) is a good starting point.

DSLs are not a new idea, they have been around forever, as illustrated by van Deursen and Klint ([Domain-specific languages: An annotated bibliography](https://www.infoq.com/articles/why-dsl-collection-anecdotes/#5)) and Section 7.5 of [Generic tools, specific languages](http://voelter.de/data/books/GenericToolsSpecificLanguages-1.0-web.pdf). However, they have been mostly used by programmers to simplify *their* tasks - the languages have been specific to *technical* domains. DSL targetted at non-programmer domain experts have become practical only in the last 10 to 15 years mainly because of better tools for developing these languages. "Better" primarily means faster. Because a DSL by definition has a smaller user base, their development must be less effort than for a general-purpose language for the business case to close. Language workbenches, tools optimized for language development, make this realistic. One particularly important feature of a language workbench is the ability to reuse (parts of) language definitions to avoid the need for redeveloping your + operator over and over again. A second enabler for DSLs is the ability to mix various notational styles to make sure the language is intuitive for the users in a wide range of domains. Again, industry-strength tools that support this have been around for maybe 10 years.

So now that we have laid the foundation, let's look at a couple of example domains and why they have successfully introduced DSLs.

**Salaries and Taxe**

**CONTEXT** Peter works for DATEV, a company that calculates payroll slips for many small and medium companies in Germany. Based on the gross salary an employee gets paid every month, the system calculates the tax they have to pay, the amount owed to the health insurance, deductions for their company car, travel reimbursements and a whole lot more. The calculations are based on lots of rules - some systematic, some rather random - resulting from decades of evolution of Germany's laws and regulations. Peter is one of the business analysts whose job it is to understand those laws and regulations and write the code that "implements" them in DATEV's system.

The software to perform those calculations has traditionally been running on the Windows PCs of DATEV's clients. But now market pressure forces DATEV to move everything into the cloud and the browser. Jochen is an architect in the department that is responsible for the technical concerns of the new system, and his team has decided to implement the new version in Java, using DDD and microservices. They have decided to rewrite the whole system because the application logic is so deeply intertwined with the specifics of the Windows implementation that it's challenging to (automatically) extract and transform it into something that runs on the cloud software stack.

The market also requires more flexibility with regards to how the calculation logic gets packaged. For example, in addition to the main "calculate everything" application used by tax consultants, DATEV wants to offer a small, self-contained web application for consumers to calculate their net salary based on the gross and some of the deductions mentioned above. Considering how much effort goes into the analysis and implementation of the calculation, they obviously don't want to implement it again and again, for every new application. Partially based on the experience of how hard it is to extract the application logic from the Windows software, Jochen realizes that it is crucial to keep the salary logic completely decoupled from the technical aspects of the system.

Peter on the other hand really doesn't care about all these technical concerns. He has studied business administration at university, and he doesn't even see himself as a programmer: "I try to understand the details of the law and systematize it in requirements documents, as carefully written prose, tables and sometimes a little bit of pseudo code." On the other hand, Peter understands that this process of expressing the calculations this way is tedious and error-prone: "It's just so inefficient. It's hard for me to be precise, I cannot really test what I write, and then the developers who are supposed to implement everything misunderstand a lot." Peter realizes that he has to become more precise, more formal, for the overall process to be more efficient, but he really doesn't want to deal with the cloud technology stack: "That would slow me down!"

Peter points out two more things: "Our domain has two major complexities that are not straight-forward to handle with the programming languages I have seen used at DATEV." The first one is that essentially all data is temporal. Consider this example that computes the churchTax for an employee's salary, their religion and a particular month:

fun churchTax(salary: real, religiousAffil: religion\_enum, month: int) {

if religiousAffil.isOneOf(catholic, protestant)

then salary \* 0.05 else 0

}

The problem here is that salary and religiousAffil aren't simple reals and enums, rather each are time series where values can change even within that month for which we calculate: an employee might get a raise on Oct 1 and then quit church on November 12.

The second problem is that the data structures and the calculation algorithms change regularly, usually every year, to reflect the evolving regulations. But DATEV is legally obliged to be able to reproduce the payroll slips for previous years. Worse, it is possible that the way you calculate something changes *within* a period over which you calculate.

Peter: "It's almost as if the program changes while it runs."

**LANGUAGE** The primary goal of DATEV's DSL is to enable Peter to do his job better. It allows him to specify the complete calculation logic and also write tests for it. He can execute these tests right in his IDE. "Just like my developer colleagues who run their unit tests this way. This really helps me build confidence in what I produce."

The language has special types and operators for currency, dates and percentages, types that are ubiquitous in his domain; the code below has examples. The language also supports tabular notations for complicated decisions as well as to collect reference data. "It's just syntax, I know, but it makes a big difference for those convoluted decision procedures expressed by German law." An example of a decision table is given in the next section.

Most importantly, however, the language has direct support for temporality and versions. The function given above would be written as follows:

rule [monthly] for ChurchTax

uses e : Employee {

result.taxValue :=

alt | e.religiousAffil!.isOneOf(catholic, protestant) => 5% of e.salary! |

| otherwise => 0 EUR |

}

Note the ! after the temporal variables religiousAffil and salary. This invokes the default reduction, an operator that automatically creates a single value from the time series. The strategy by which this single value is created is given as part of the definition of the data structure (which is not shown here). For the religious affiliation, it uses the last value in the given period (as the law says!) and for the salary, it uses the weighted average. Apropos "given period": because the rule is marked as monthly there is an implicit parameter period that is used with the default reductions. Peter: "I can write code that looks very intuitive, and the temporal stuff happens below the surface."

Let's look at versioning. The language supports the explicit declaration of versions in the code. Data structures and rules (such as the one above) can be overridden in a new version. The well-formedness of the new version relative to the previous one is checked by the (domain-specific) static type system, "with error messages I can make sense of", says Peter. The runtime then performs a kind of polymorphic dispatch over the calculation rules for different points in time, even if you, for example, calculate this church tax for all 12 months of a year, and the rule changes mid-year.

"That's pretty neat", Peter says. "Even though there was a bit of a learning curve for me, to work with a language. But the IDE, the understandable error messages and the integrated visualizations of data relationships and versions really helps. It's worth it!"

And Jochen the architect? "Well, I really like the approach. For one, I get fully tested application logic. Those guys even have a coverage analyzer! In fact, I really don't even look at it anymore, I just run the [generator that produces a JAR file that I plug](https://medium.com/@markusvoelter/the-business-dsl-devops-process-fb5531bf1f50) into my Spring service." The fact that the application logic is independent of the particular implementation in Java is useful as well: "We've generated parts of it into JavaScript for validation in the browser." And even during the development of the language, the team changed the stack from JEE to Spring, "and it was very easy to retarget the application logic." What about the effort to build all of this? Jochen again: "My colleagues who built the language and the runtime and the generators struggled a bit at the beginning with all the language technology, but eventually, they got the hang of it."

**LEARNINGS** Separating application logic from the technology is a well-known reason to use a DSL, but it's worth revisiting. Why was it so hard for DATEV to extract the domain functionality from the legacy system? So hard in fact that they manually reverse engineered everything and wrote it all again from scratch? Because once you've encoded business logic in any general-purpose language, so much of the domain semantics is lost that it is very very hard to reverse-engineer it automatically. Sure, you can potentially translate Java to Kotlin or whatever, but you won't "recover" the clean domain semantics. It is therefore absolutely crucial to separate business from technology and encode the domain semantics as [cleanly](https://medium.com/@markusvoelter/funclerative-programming-968cc53ee580)) as [possible](https://languageengineering.io/an-argument-for-the-isolation-of-fachlichkeit-3a67a939d23b). This is *the* asset for your business! DSLs are best suited to realizing this ideal situation.

It is also easy to see that tables and other non-linear notations are useful for expressing complex decisions. What is not quite as obvious is how important the DSL is in helping with the [hidden complexities of the domain](https://medium.com/@markusvoelter/the-hidden-complexities-in-fachlichkeit-23884a4d8d0a), temporal data and versioning in this case. It took a little while for the development team to dig up that these are one major source of complexity and to find good solutions to address them.

Developer lore also suggests that domain experts are happy to express themselves imprecisely in Word (and then blame the developers if stuff goes wrong later). While such folks exist (I have met them), there are also many like Peter who see the shortcoming of a document-based approach and are willing to learn if their needs are taken seriously - and, for example, embodied in a DSL.

**STUMBLING BLOCKS** Business programmers, at least DATEV's, care much more about details - such as names of keywords - than we thought we underestimated the effort of taking care of these. They also value orthogonality and conceptual consistency less, which means that the implementation of the runtimes becomes more work. Some of the especially fancy approaches to versioning were so hard to understand, even with tool support, that we replaced them by less flexible, but easier-to-understand alternatives (to the great frustration of the language designers who came up with the more elegant solutions).

**SIMILAR CASES** We have built several other DSLs in business domains that are guided by the same success criteria. One is to calculate the yearly tax declaration for German citizens, also for DATEV. The language also uses temporal data, but does not need the same support for versioning, because each calendar year is separate, and branching in git is good enough. The top-level structure is basically a big tree that resembles the terminological hierarchy of German tax law. The language also supports implicit monthly calculations as well as the declarative specification of optimizations. Tests get run with an interpreter, and for final deployment, the model gets generated to C (for the on-premise legacy app) and to Java (for the cloud solution). Interestingly, the execution model relies on incremental computation based on statically analyzed dependencies in the calculation tree. The system replaces an older imperative DSL that had been used for two decades.

We recently received this very nice email in which the department head commented on a video in which Julia had reviewed a model created by Peter:

Hello Julia, Peter gave me your video in which you reviewed his property valuation model. I'm thrilled with the constructive way you commented, and it fascinates me how you can apparently understand what Peter is expressing there ...without reading the law itself.

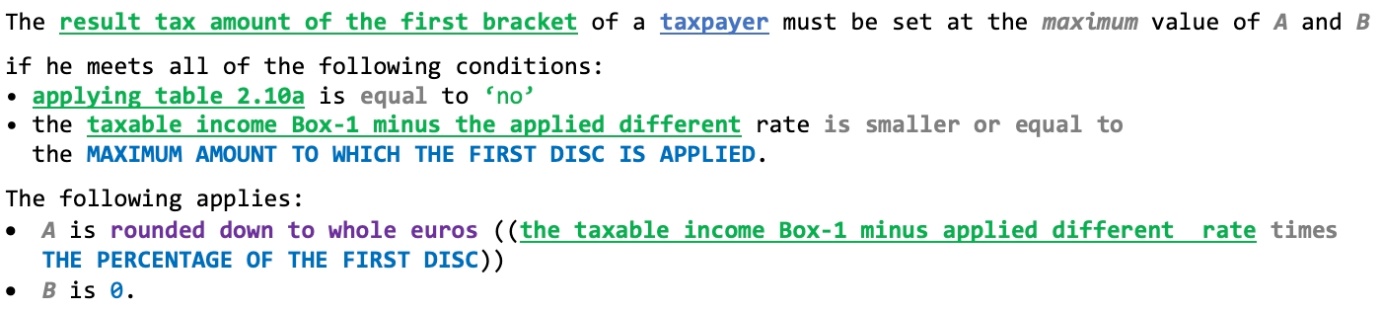
Although I've seen very little of this language so far, I think this video gives me a pretty good impression of what makes the language tick, especially the hierarchical approach aligned with the law. It was actually fun to try to keep up with your thought, and see how you refactored the hierarchy to better align with the text of the law.

In addition to the law-aligned hierarchy, what I really like is the possibility to explicitly describe law-related data values and, if necessary, even use tables. It is also interesting to see how much functionality the editor provides when you move whole subtrees, which checks it performs, and how you can find out which values are used where.

I understand now what makes you so enthusiastic about the language, and I really felt the urge to develop something with it myself.

This feedback says it all.

The regulations that determine how governments collect taxes and pay out benefits are certainly complex enough to warrant the use of DSLs. For example, in the Netherlands, the public benefits payments get calculated by a system whose calculation core gets generated from DSL-based models. The Dutch Tax Agency has a history of modeling business rules and has moved everything to DSLs and code generators over the last years. They are now working on doing the same for the tax system. To this end, they have built the Agile Law Execution Factory, or [ALEF](https://resources.jetbrains.com/storage/products/mps/docs/MPS_DTO_Case_Study.pdf). The languages rely heavily on prose-like syntax as well tests specified by the end-user and a debugger that overlays the values of properties over the source code.



To see improvements brought about by the DSL, it is useful to contrast the contents of a phone call the tax agency would get from the government when the government were planning a change to the tax or benefits law. In pre-DSL times, the call would go something like this: "If we made this change to the law, would you able to implement that change within the next 12 months?" They wanted to ensure the change is actually implementable. Now, with the DSL-based approach, the call has been overheard to be quite different: "Hey, we're thinking about making this change in the law, can you please modify your rules quickly and tell us what the impact would be?" This is clear testament to the increase in agility and the reduction of effort in programming the tax and benefits systems.

A final example is in the insurance domain. In this case, the customers also relied on Word-based specifications of the calculation logic of insurance products, with subsequent manual implementation in C. However, in this case, the Word documents contained formal specifications: they used a document template that each specified a function model, that combined documentation, parameter declarations, and the actual behavior. For the latter, the customer invented an imperative programming language which the insurance programmers actually typed into Word; they were effectively pen & paper programmers. Because of the lack of IDE support, and the ability to write tests, these specifications were of not-so-great quality. The goal of the DSL was to build a real language, with type checking and IDE support, and automatically generate the C code. The syntax of the language was to be reused, and the overall look of the Word documents should be retained as well. The language has been used in production for two years now.

**Algorithms in Healthcare**

**CONTEXT** Digital therapeutics is a field of healthcare in which digital technologies, mostly phones, are used to diagnose or treat medical conditions. It is applied in behavioral therapy (where an app might get used as a diary or guide the patient in performing certain behaviors), to help diagnose diseases (for example, in the context of sleep or nutrition) and to administer drugs (for example, those that manage the side-effects in the context of chemotherapy).

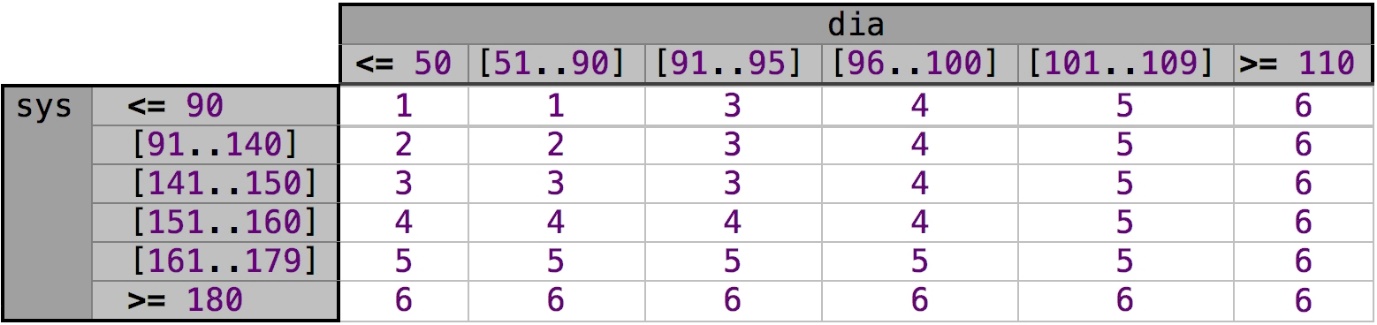
Voluntis is a French-American company that develops such apps and their respective server-side backends. Like DATEV, they are forced to reduce their time-to-market and increase the variety of apps they release to be successful in the dynamic market environment characteristic of e-health.

And just like in the previous example from the payroll domain, it is obvious how separating the technical implementation of an app from the algorithmic aspects of the diagnosis or treatment is useful: the former gets done by software engineers, the latter by healthcare professionals. And similar to the business programmers at DATEV, these healthcare professionals do not consider themselves programmers. Adrienne, an engineer at Voluntis, once summarized it like this: "Doctors don't know what a function call is!".

There's also a challenge with regards to platform independence. The apps must run on both iOS and Android, and some parts get executed in the browser or the server. So it is desirable that the core algorithm is completely decoupled from the implementation technology. This is especially important because of a particular challenge with app development: to conserve battery power, mobile operating systems limit what an app can do in the background. In particular, the kind asynchronous behaviors many of these algorithms require must make creative use of reminders and push notifications, and these work quite differently on iOS and Android.

But the main driver for looking into a DSL for Voluntis was the regulatory context. Adrienne: "Our apps are software medical devices, which means that they are subject to regulatory oversight from the FDA or its international counterparts." Exactly what this means depends on what the app does - a diary-style app is regulated less stringently than one that calculates drug doses or potentially even controls a mechanical device such as an insulin pump - but all require extensive documentation of requirements, design and test. Martin is a member of the validation team at Voluntis, and he reports that " a significant part of the development cost relates to this paperwork." It is well known that the cost of fixing an error rises the later in the development cycle you find it; obviously this cost rises dramatically in a highly regulated context. Martin: "If we find a bug in the implementation, we have to redo lots of reviews, rerun validation activities and update several reports." So finding problems early, ideally during the requirements and design phases would be extremely beneficial. Martin: "Finding such problems in the medical literature written by the docs is very hard." Adrienne adds: "When we write the code, this is the first time the algorithms are formalized and can be executed. That's when we find problems such as missing branches or boundary cases." And Martin again: "Same for us. When we try to validate whether the code implements the requirements correctly, we have a hard time understanding the expected behavior from the literature."

**LANGUAGE** Voluntis solution uses three tightly-integrated DSLs. The first one is a functional language used for encoding decisions and calculations. Many decisions are represented as decision tables or decision trees. Medical decision procedures are often based on empirical research or based on heuristics; they cannot necessarily be described by a simple formula. Adrienne adds: "Tables and decision trees are also found in the medical literature and in studies as well, so doctors are used to them." Tables and trees probably sound familiar from the payroll example above. And indeed, the tables and decision trees are similar in both languages; both DSLs build on the same extensible functional core language called [KernelF](https://www.infoq.com/articles/why-dsl-collection-anecdotes/" \l "13).



The second language is used to encode the interactive algorithm that prompts the patient for input, delegates to the functional language to make decisions or calculate parameters, and then reminds the user of various actions at particular times. This language is based on state machines, a formalism that is suitable for asynchronous, reactive behaviours. But the DSL adds to generic state machines significantly: for example, it can declaratively express timeouts (*in X hours do this*), specify behavior that repeats in time (*every day at 8 do this*), express linear sequences of actions that can be "interrupted" and separate the typical flow of actions from cases that represent complications.

The third language is used for testing these algorithms. Users can script interactions of the patient with the app and assert specific behaviors. The tests can be executed in the IDE using an interpreter. There's also a simulator that runs the app with a simplified phone UI to let medical personnel interactively validate the algorithm at an early stage. The simulator can be used to record executions and then translate them into a test case for downstream automated regression testing.

Christine is one of the medical experts who works for Voluntis on the algorithms in the apps. She admits that "it took some time to learn the language and appreciate the value of what the developers call test-driven development," but then concludes that "it really pays off. It's really nice how we can make the algo come to life literally as we design it, interactively." She also remembers a story that she was particularly impressed with: "We ran several rounds of refinement of a blood pressure algorithm with oncologists from a partner hospital. The goal was to jointly define thresholds to trigger medical recommendations for a patient to call their medical team. Using the DSL and the simulator, we were able to define and validate the requirements within a few days. In the past, based on documents, something like this could take weeks or months."

This approach also has lots of advantages for Martin and his colleagues when they validate the algorithm: "The models are much more precise than the documents we used earlier. And they are higher level than source code and free of technical stuff, so it's actually feasible to review them to understand the algo. And we can experience them through the simulator." In addition, the system measures coverage of tests, supports tracing to high-level textual requirements and automatically generates many of the reports required for FDA approval. Martin says, only half-jokingly, that "initially, we worried that we'd all be fired because the creation of all these documents is automated to a large degree. But instead we now have more time to think about the validation approach and how to automate it. In the end, we are able to deliver more products to the market, which is all good." Martin illustrates the improvements with this story: "If a change to the algo is necessary after verification/validation, large parts of the verification/validation have to be repeated. With the old approach, for each set of changes, the testers had to go through all existing tests, potentially adapt them, and then reexecute all. This took up to 10 days. Now it's done in a few hours for tests review/update, and a few minutes for tests execution with the interpreter. Sometimes up to 5 of these change/verify/validate rounds are necessary. The overall improvements are tremendous."

Of course we had to make sure that the runtime infrastructure did not introduce additional risks (in the sense that a correctly validated model is executed in the wrong way). We used redundancy in the architecture and the development of the runtime and generators plus coverage measurement and manual reviews of some parts of the infrastructure to ensure the necessary quality([Using language workbenches and domain-specic languages for safety-critical software development](https://www.infoq.com/articles/why-dsl-collection-anecdotes/#10)). Adrienne summarizes, with pride in her voice: "It is a testament to the level of quality achievable with this approach that very few faults were discovered in the algorithm's execution, and all were fixed very early in the development. And the first app developed using the DSLs had no fault after it went to market!"

**LEARNINGS** The simulator and debugger were crucial. The less technical the DSL users are, the more important it is that they can play with the language, and not just "stare at models", no matter how high-level they are.

Contrary to folklore, it is absolutely possible to use a custom language with a custom runtime in a safety-critical context. You are *not* forced to use one of the (few and expensive) qualified model-driven development tools. Of course you have to put in place the right architecture to ensure quality and reliability, but this is certainly feasible.

We have seen how the payroll example and the decision/calculation parts of the language discussed here relied on functional programming; and the main behavioral part of the DSL here is basically a state machine. It is generally a good idea to rely on existing, proven and well-understood execution paradigms - and then extend them with domain-specific, higher-level abstractions - instead of trying to invent your own.

Reuse works beyond the conceptual. The ability to reuse parts of languages - KernelF in this case, and in the case of DATEV - is really helpful, for the same reasons why code reuse in the form of frameworks and libraries is also good: it works, it's tested and you just don't have to spend the effort to develop it yourself. However, reusing languages can be technically difficult, so make sure you chose an implementation technology that is able to reuse languages effectively (see [Language and IDE Development, Modularization and Composition with MPS](https://www.infoq.com/articles/why-dsl-collection-anecdotes/#6) for more details).

**STUMBLING BLOCKS** We had planned to integrate model checking and SMT solving to verify certain correctness properties of the DSL programs statically (as opposed to uncovering problems via tests). However, it proved really hard to do this for a language which, at the time, was already relatively big. A smarter approach to verification-driven modeling is to do it bottom up.

An initial version of the language used the state machine paradigm without many of the necessary domain-specific extensions. This led to relatively verbose models that were hard to read, compromising the acceptance by Christine and her colleagues. It took a while to dig out of this reputational hole again.

The quality assurance people (represented by Martin in this anecdote) were quite a bit more skeptical initially. They stuck much more to the traditional means of validation - an understandable tendency, since they are ultimately *responsible* for patient safety (and for business success in the sense that they have to get the paperwork through the FDA). But once they were given more time to learn and experience the new approach, they came around.

**Spacecraft Software**

**CONTEXT** OHB is one of the leading manufacturers of satellites in Europe. For example, OHB has been developing and is [currently manufacturing](https://www.ohb.de/de/tag/td/galileo-1/) all 34 satellites for the European satellite navigation system [Galileo](https://en.wikipedia.org/wiki/Galileo_(satellite_navigation)). They are also involved in the third generation of Meteosat weather satellites.

At a high level, the software for a satellite can be split between the space segment (the software that runs on the satellite) and the ground segment (the software used for configuring, monitoring and controlling the satellite from the ground). The space segment can be separated into the attitude control algorithm and the so-called on-board software. Attitude control is a classical continuous control loop, developed with Matlab/Simulink, that continuously maintains the correct attitude, where "correct" is determined by various concerns such as pointing the sensors in towards an interesting target, orienting the engines to align their thrust vector, and maintaining thermal balance by exposing different parts of the satellite to the sun and/or coldness of space. The on-board software manages everything else on the satellite and is a discrete program, typically implemented in C. Ralph is one of the engineers responsible for the on-board software at OHB: "It's basically typical embedded software, where timeliness and careful use of memory are a concern."

Many parts of the software are effectively state machines. One example is global mode management: the satellite might be in operation mode in which, for example, Meteosat, satellite uses its sensors to collect data; it might be in downlink mode during which it transmits data to an antenna on Earth as it flies over that station; or it might be in maintenance mode, where new software gets uploaded or parameters are changed. As these modes change, the functionalities of various subsystems have to be coordinated.

Ralph explains the basic structure of a satellite: "A satellite can be split into two parts: the payload and the bus. The payload is what the operator earns money with: sensors that observe the Earth or atomic clocks and antennas that create and transmit the GPS signal. The bus is the backbone infrastructure that's necessary to keep that payload on a stable orbit around Earth and supply all the resources it needs to do its work. It deals with attitude determination and control, power management, thermal management, command and monitoring and data downlink." While the software that controls the payload is custom developed for a satellite's particular payload, there is a lot of opportunity for reuse for those parts that control the bus - in fact, satellite vendors have bus "platforms" that they customize for particular applications.

An important aspect of satellite software is that many of its functionalities must be controllable remotely, from the ground, via some kind of radio network protocol: parameter values must be updated, commands must be executed and lots of data values must be monitored. A corresponding module in the ground segment must know about these parameters and commands and their encoding in the protocol and expose it to the operator via some kind of UI.

From this discussion it becomes clear that some kind of componentized architecture makes sense. Ralph: "Reusable components for controlling the bus services plus custom components for running the payload get combined into a common runtime infrastructure that deals with global resource management, mode switching and remote communication."

In fact, one of OHB's main customers, the European Space Agency, has a lot of standards for such on-board software. It embraces the notion of components, but also prescribes a lot of details, for example, how to deal with the highly redundant space-qualified processor board. OHB and its competitors are required to develop their software in compliance with this standard, and, as part of the deliverables, they have to document their compliance. Plus, they have to fulfil lots of other requirements regarding software quality. Ralph recaps a conversation with someone named Adrienne, who works in the healthcare industry; they met at a software conference: "In that sense, this software is regulated quite similar to software in the medical domain. And we have some of the same problems: it is a lot of tedious work to keep the documentation in sync with the code, we all hate this aspect of our job. The fact that we have to use traditional UML tools to maintain design models doesn't make it any better."

**LANGUAGE** A few years ago Ralph decided to build a DSL specific to satellite on-board software based on mbeddr ([mbeddr: Instantiating a language workbench in the embedded software domain](https://www.infoq.com/articles/why-dsl-collection-anecdotes/" \l "12)). mbeddr is a full implementation of the C programming language on top of a language workbench. This means that the language engineering capabilities of the language workbench can be used to extend C with domain-specific concepts. Ralph: "The key idea was to take the abstractions prescribed by the applicable communication standard and the ESA-adviced architecture guidelines and represent them directly as language constructs. Because these things are now *part of the source code* they are correct-by-construction and always in sync with the code, in contrast to the UML models we used before. The code is also much easier to read, especially if you know the standard. It's really quite neat!" OHB's 'dialect' of C has components, the notion of modes, as well as commands that can be triggered from the ground and attributes that can be read via telemetry. It also has means of defining the encoding of these commands and attributes on top of the telemetry protocols.

mbeddr is actually not just a complete implementation of C, it comes with a set of reusable extensions useful for embedded software. These extensions are modular, so a user can choose which of them they might use in their system. You import a language extension just like you import a library in a classical setting, but you get IDE support, type checking and static translation as well. Ralph: "Two of mbeddr's extensions were especially useful for us. One was the state machines, because lots of behavior in our system can be nicely modeled this way. The other one were physical units, where a variable can have a unit in addition to its numerical type, and the IDE performs unit checking and conversion. Since our software deals with a physical object, the satellite, most of the quantities we deal with in our code naturally have a unit."

mbeddr also comes with an extension for interfaces/components/connectors, but OHB chose to not use them and instead implement the slightly different component abstraction required by the ESA standard.

Because the abstractions available in the DSL have been chosen to correspond to those required by ESA, the majority of the required design documentation can be generated directly from this source code. This includes UML class diagrams, sequence diagrams and state diagrams. Because they are generated *from the source code* they are always up to date; and because the code contains domain-relevant abstractions, *they are meaningful to the reader*. Ralph: "Actually you can do code review on paper, if you are so inclined." An additional generated artifact is the software for the part of the ground station that is used to command the satellite, at least a prototypical one: because everything that is accessible via telemetry is defined explicitly as such in the code, a simple GUI to "play" with the satellite can be generated. The same is true for the code that maps those commands and attributes to the telemetry protocol: automatically generated as well.

Ralph summarizes: "You see, the actual behavior implemented by this software is not terribly complicated. A satellite doesn't do that much. What drives the implementation effort is the fact that everything has to conform to those standards, that we have to deal with redundancy and robustness mechanisms, that basically everything must be remotely accessible and that we have to prepare lots of paperwork for our customer. The DSL and the generators automate almost all of that. This is why I have told my bosses that if we'd use this approach in the future, I can be 10 to 25 times faster than with the traditional approach. The effort for building the languages will amortize even for the first satellite we build this way!"

What about the point we started with, that embedded software must be careful about performance and resource consumption? Ralph: "Our transformations map the DSL abstractions to the same low-level C constructs that we had written manually before, so we do not see any significant overhead." And some of mbeddr's extensions, for example the physical units, are relevant only on model level for type checking and do not have any representation in the generated C code (except value conversions, e.g., multiplying by 1,000 when converting from km to m, but you'd have that in manually written code as well).

**LEARNINGS** We've started this paper with the classical business domains, where the DSL users are *not* programmers; the closeness of the abstractions to the domain and the use of readable notations was crucial to get buy-in from the users. In the healthcare domain those concerns were relevant as well, but safety came in as an additional concern. In the domain of spaceflight software, the users of the DSLs are software engineers, so they are used to programming. Here the challenge to adoption is about trusting the generator, the feeling of loss of control over what happens in detail and the fears about performance and memory use. The primary reason for using a DSL is the significant increase in developer productivity through automatic generation of boilerplate code and the implicit consistency between the code and documentation artifacts.

The spaceflight DSL is also an example of language extension, where a full-blown programming language - C in this case - is extended incrementally to support more and more domain specific concepts, first within mbeddr itself (units, state machines) and then specifically for ESA-compliant on-board software. The previous examples in this article also reused an existing language - the functional language KernelF - but it was embedded into a completely new language; for users, it doesn't feel like language extension.

And last but not least: a developer suggesting to management that they can implement a system with 10 or 20 times (not percent!) less effort is a big deal. In fact, this 10-fold increase qualifies as a silver bullet, which according to Brooks seminal work [No Silver Bullet](https://www.infoq.com/articles/why-dsl-collection-anecdotes/#2) doesn't exist.

**STUMBLING BLOCKS** Since space business traditionally is a very conservative domain and technological advances are adopted only very slowly, the DSL-based approach has not made it into production (yet). However, the language has been implemented to a substantial degree, and a significant part of a typical satellite software has been implemented as part of a research project; this is where Ralph came up with the 10 - 25 times faster claim.

Admittedly, adopting such an approach is a big change. And change always implies risk. But on the other hand, we're talking about a huge improvement in productivity. Organizational and "political" resistance against the approach is generally a major reason why this DSL-based future is so unevenly distributed.

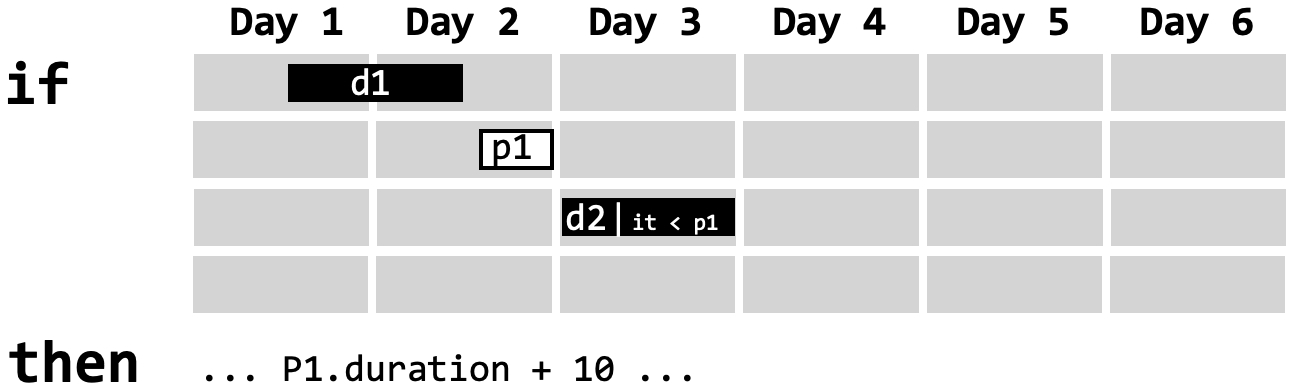
**SIMILAR CASES** Shortly after the development of mbeddr itself, a couple of cooleagues at itemis developed a smart meter([Using c language extensions for developing embedded software: a case study](https://www.infoq.com/articles/why-dsl-collection-anecdotes/#9)), one these IOT-style connected devices that measures a building's power consumption. They used mbeddr's components to modularize the overall system: the hardware access layer, communication within and beyond the device, the current and voltage measurement subsystems plus the application layer that calculated customer-relevant statistics and drove the display. Many of these modular components were reused later in other devices. The memory and performance overhead induced by the nice structure was so low that the system still ran on the hardware mandated by the customer, on the order of a few percent. The team also made use of state machine and units - obviously useful when you deal with a whole range of electrical quantities.

The team also developed several extensions to mbeddr C: one is for defining message structures for inter-processor communication (the system had two MSP430 processors), one for the low-level metrology and one for the higher-level application. Another provides a more convenient and more type-safe syntax for working with registers and interrupts. Such low-level work is necessary to achieve the 4,096 Hz sample rate necessary for precisely measuring current and voltage. Despite the "overhead" of a DSL, the precision achieved by this measurement was better than what was required by the spec!

Even though this was the first use of mbeddr in a real project, and the team even developed a bunch of mbeddr extensions along the way, the project was finished roughly on time and budget; a detailed discussion of this case study is available [in Using C language extensions for developing embedded software: a case study](https://www.infoq.com/articles/why-dsl-collection-anecdotes/#9).

Another DSL in the space of embedded software is worth mentioning. It concerns tachometers, the devices in trucks that track driving times and break times. These days they are digital, of course, and they don't just log time. They actually know about the legal requirements and constraints and proactively tell the driver when they have to take a break, and for how long. And as you would expect, the rules that govern admissble driving and required break times are complicated, there are multiple dozens of rules that handle various special cases. And they are all slightly different for each country in Europe, the market of our customer. Getting these rules right is a major challenge for our customer who builds these devices.

Our customer developed a DSL that uses a tabular notation (inspired and prototyped in Excel) to express these rules. We implemented this notation as an actual language with a supporting IDE and type checking. Users could also specify example scenarios and express assertions for testing. All of this was then generated to C code (by translating to mbeddr) for direct execution on the actual tachometer hardware. Note how this example combines a specification language that is intended for an analyst (the person who reads the laws and understands the rules) but then still generates to an embedded system.



**Models for Analysis**

So far, the models and languages we have discussed in this article were primarily intended for execution through code generation or interpretation. In this sense, the models were programs. Let's now look at cases where DSLs are used to describe models that are intended for various forms of analysis. Meet [Dan](https://sites.google.com/site/ratiud/home) ), a software engineer who works as an internal consultant for UFL, a large engineering company.

**CONTEXT** Dan specializes in formal verification, a set of mathematical approaches to proof certain properties about models. Formal verification is similar to testing in that it can find faults in software. But in contrast to testing, a verification typically does not run the program for one particular set of inputs, but explores a whole set of program executions for a range of inputs. Dan summarizes: "In this sense, verification provides stronger guarantees, it can potentially find more faults. Some verification approaches can also proof the absence of errors, something that testing can by definition never achieve for realistically-sized systems."

But of course there is no free lunch, and using verification instead of testing also has its challenges. Dan: "Verification relies on quite sophisticated algorithms and math, and often this leaks through to the user. You have to describe your system in a formalism that fits the kind of property you want to verify, and you also have to express the property itself." Both are typically expressed with often quite low-level specification languages. "Requiring engineers to express their systems in these formalisms and finding suitable formulations of the properties they want to show -- an example would be AG(f4 -> (!e4 AW e3)) -- is really an uphill struggle," says Dan. "In addition, many of front-end tools, you know, what they call IDEs, are not up to snuff compared to the engineering tools people are used to. They have been traditionally targeted towards verifying critical properties of dependable systems in projects with big budgets that can afford highly specialized verification experts. The advances of verification engines from the last 10+ years make it feasible to use verification in more mainstream projects, but better tool usability is urgently needed!"

A verification technique that is particularly useful in practice is model checking. It allows the verification of stateful, discrete behaviours, typically expressed as state machines. The canonical example is traffic lights. Imagine a multiway crossing, where each incoming road has a set of red/yellow/green lights. For the crossing to be safe, you have to guarantee that the software never switches two incoming roads to green at the same time. Similarly, for the system to be fair, you have to guarantee that every incoming road gets a green light eventually. Demonstrating that a software exhibits such safety and liveness properties, for all possible executions, is the goal of model checking.

Dan's collaborator Peter chimes in: "Our company builds lots of safety-critical systems in the transportation, energy and healthcare sector. To guarantee safety, we have to make sure the logic works as expected, as Dan has just explained. But in addition, we also have to add architectural means that increase safety, for example a redundant controller that takes over when the primary one fails. However, the necessary failover logic is non-trivial and comes with its own risks and potential for errors: we have to guarantee that only one of the controllers is active at a time, that the two don't interfere. That is really quite non-trivial. And we absolutely cannot affort that all of this safety-assurance is done by a small number of 'verification gods' whom everybody has to trust blindly. We have to make this accessible for the majority of our engineers."

**LANGUAGE** To tackle this challenge, Dan has developed [FASTEN](https://sites.google.com/site/fastenroot/), an integrated tool that supports various verification approaches, including model checking([Fasten: An open extensible framework to experiment with formal specication approaches](https://www.infoq.com/articles/why-dsl-collection-anecdotes/#4)). The actual model checking is done by the NuSMV and Spin model checkers, both are acclaimed for their algorithmic power. As the starting point, Dan has implemented NuSMV's SMV modeling language in FASTEN: "Every NuSMV model is also a valid FASTEN model."

One level up in convenience, FASTEN supports the graphical definition of state machines. While this does not add to expressivity, it makes the models easier to digest for many users. The first real increase in abstraction is the addition of typedefs and structs, both helping to make the model more readable -- native SMV just supports primitive types. A more meaningful addition is the notion of generalized test cases. In regular testing, the user specifies a specific value for each input of a program, and asserts over all outputs. FASTEN's generalized test cases allow the user to use value ranges in the inputs. Dan: "This exploits the underlying model checker's ability to explore all execution traces of a program instead of just 'running' the program for one set of inputs". Peter adds: "This is nice, because we can stick with the familiar notion of a test, but we still benefit from the verification capabilities of the model checker."

If the model checker finds a program execution for which a property does not hold, this is reported as a so-called counterexample: it is a hint for the developer to "fix" the model because the program does not conform to the property ascribed to it. Understanding the counterexample is not always easy, especially when it represents forgotten corner cases. Dan adds, with a grin, "model checkers are quite pedantic and identify *all* missing cases, regardless of their perceived relevance." One thing FASTEN does to make understanding counterexamples easier is to lift it to the abstraction level of the user's input language, state machines or test cases. In addition, the user can interactively step through the counterexample, almost like a debugger in an imperative language. Peter: "Another benefitial similarity to testing."

As systems get larger, modularization becomes crucial: a module with a clearly-defined interface allows users to reason about the module in isolation. The overall system is then assembled from hierarchically composed modules. This hierarchical decomposition is also useful in the context of verification, because it allows the verifier to modularize its exploration of the set of all possible traces through the model. Long story short, another abstraction available to FASTEN users is components and interfaces. The interface definitions include the specification of assumptions and guarantees that define the behavior of the component as seen through the interface. When systems are assembled from such components, two checks become possible: (1) the implementation of a component A that interacts with a component B through its interface can be checked for compatibility with this interface B; and (2) component B's implementation can be checked against the behavior specified by its own interface. The overall verification of the system relies on separately verifying all such combinations. Peter again: "This kind of modularization is crucial to be able to reason about realistic systems such as our power steering unit. The abstractions provided by FASTEN make this approach much easier compared to coding everything manually in low-level NuSMV."

A final extension available in FASTEN concerns the specification of properties. Because these properties make statements about sets of program executions, they can be quite non-trivial; they quantify both over the structure of a single trace (*in all future steps ...* or *in some future step ...*) and about sets of traces (*for all traces ...* or *there exists a trace ...*). To simplify the creation of such properties, FASTEN provides a set of predefined patterns for common cases. They are easier to write and read. And like with all incremental language extensions, users can always "step down" to the lower level of abstraction for less common cases, without changing the tool environment. Peter: "I can also build additional patterns as I need them, we've done that a few times".

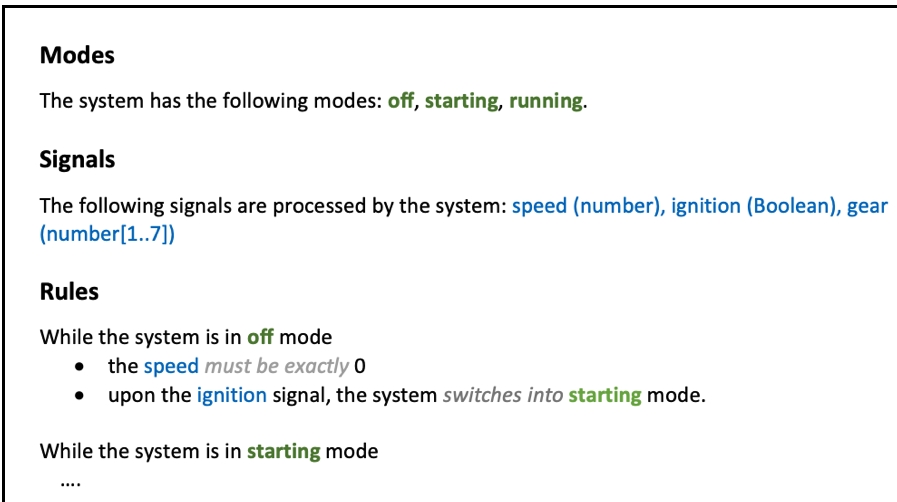
**LEARNINGS** The (usually academic) developers of the model checkers focus on the algorithms: the scope of models they can process, the range of properties they can check, and the speed and memory consumption associated with these checks, especially as models become larger. An engineer (usually in industry) assumes all of these things to "just work", like compilers. Instead they focus on robustness, ease of use and IDE support. Even more, engineers don't just want to run a verification, they want to build complete safety cases. This is why FASTEN also includes languages for requirements and formalized safety arguments. Similarly, the algorithm developers tend to use a generic (and hence, low-level) input language -- one that is close to the mathematics behind the verification -- to make the tool useful for a wide range of use cases. Engineers prefer modeling formalisms that are tailored to their particular context. Using a language workbench that allows modular language extension of the low-level input language is a really approach for addressing both needs.

**STUMBLING BLOCKS** Translating abstract models down to the low-level input language of NuSMV -- potentially over multiple levels -- is not that hard. Lifting the low-level counterexample up to the abstraction level the user cares about is much harder and was a lot of work for Dan when he built FASTEN.

The other issue with the approach of abstracting from the details of formal methods -- model checking in this case -- is that the abstraction is sometimes leaky. For example, bounded model checkers like NuSMV have to be parametrized regarding how exactly they run the check. One example of such a configuration is how often they should "run around" in a loop that is not bounded by the model (a simple counter is an example of such an unbounded variable). They cannot iterate forever, since then the analysis would never terminate. But whenever you bound the analysis to some number N, the checker might possibly find a counterexample when inspecting iteration N + 1. There are symbolic model checkers that do not have this problem, but those have other abstraction leaks. So some understanding of the model checking procedure by the end user is necessary, no matter the abstractions.

**SIMILAR CASES** Once a system is modeled as a state machine, lots of interesting things can be done with the model: it can be run interactively to allow users to "play" with the functionality expressed by the machine, users can annotate it with invariants (assumptions about the system that must always hold), users can write test cases, and, last but certainly not least, engineers can use model checking to "proof" that certain properties of the machine hold for all possible executions (or let the checker come up with a counter example).

However, during early phases of a design, certain stakeholders (some of them quite technical) expect the design to be expressed as prose text, *to make sure anybody can read it*. The next figure shows an example of a specification regarding mode management, something very common in many technical systems. This behavior quite obviously resembles a state machine. During the early specification phases, engineers (at least those at our rather large customer) wrote such documents in Word -- without any tool support for even structural consistency checking let alone verification.



It should not come as a big surprise that we have built a DSL that uses a prose-oriented syntax to "write a specification" while, in the background, a state machine is constructed. The most obvious benefit of this approach is the structural consistency: you can't mention a signal that isn't declared. Type checking in expressions is also supported. And everything else you can do with state machines -- running, testing, model checking -- is supported based on this "document" notation.

Both of the examples above cover languages and tools for verifying discrete behavior, based on state machines. They use model checking as the verification formalism. This final example adresses performance in the face of constrained resources based on a tool called Simbench([Resource analysis of automotive/infotainment systems based on domain-specic models](https://www.infoq.com/articles/why-dsl-collection-anecdotes/#1)). Imagine an automotive navigation system; driver experience requires that it be up and running in one to two seconds after the [driver turns the ignition](https://www.eetimes.com/fast-booting-techniques-meet-automotive-infotainment-telematics-activation-needs/). To be up and running, the computer inside the navigation system has to run through its boot sequence, starting processes, initializing devices and loading data. Considering the limited bandwidth of flash storage and memory buses as well as the finite speed of the processor, the question is: in which order do we start up the processes so that overall startup time is minimal because all resources are used optimally all the time and processes don't contend over those resources? And how much of each resource do we need? Figuring this out during early phases of the system design is crucial, because it is very expensive to add more hardware resources *after* the hardware manufacturing has begun.

To perform this analysis, the user first models the various components that make up the system. This includes both resources (such as processors, buses and storage) as well as consumers (primarily software processes). The various kinds of resources specify their capacity (for example, bandwidth for a bus or storage device or GFlops for the CPU). The processes are modeled as a sequence of steps, where each step specifies how much of which resource it needs (40% of a processor's performance or 5 of 50 MB/s of storage bandwidth) and for how long. A step can also have a dependency on the result of steps of other processes. Ultimately, the processes with their steps define a graph of activities, each requiring a shares of several resources.

The analysis then simulates the evolution of time. For each point in time, the degree of utilization of each resource is calculated. If a resource is used by more than one component at the same time, they have to share the resource. This is simulated by a simple scheduling algorithm that allocates equal shares for each process. Shared access to a resource means that the using processes take longer. In addition, a step is also blocked if a dependency has not completed yet. The simulation runs until all processes have completed all their steps. The system reports the running time (remember, the use case was to simulate how long a startup process might take) and illustrates resource contentions and blocking dependencies through various diagrams. Users can also write requirements as semi-prose text, such as the system should not use more than 80% of main memory at any time or the init process should be finished by 200 ms; Simbench verifies these requirements during the simulation and points out which are met and which are violated. Ultimately, the user can change the model (steps, their sequencing and dependencies, available resources) to optimize the overall process.

Two things are particularly beautiful. First, because the scheme assumes that resource usage is constant between the beginning and end of a step with some duration, the analysis can be implemented as a discrete event simulation; this means that only the times when things *change* have to be "executed", the simulation can jump to the next of these discrete time steps right away. This means that even big scenarios run in milliseconds on a normal PC. This is short enough for interactive execution as part of the type checker. In other words: the analysis results are available immediately after a change to the model. Turnaround is instantaneous like in Excel: what the user sees in the model always represents the "execution state".

The second beautiful aspect refers to the scenario descriptions themselves. Where do those time steps and resource usages (step X needs 30% of resource Y for 20 ms) come from? During early design, when neither hardware nor software exists, these are estimates based on past experience of the engineers or other, similar systems. As the design progresses, and, for example, the size of the map data becomes known, the guesses can be replaced by actual numbers. Over time, as more details of the system become available, more guesses can be replaced with factual data and measurements. This way the analysis will become more and more precise over time.

**Closing Thoughts**

*There is no silver bullet.*  
Fred Brooks

In his 1986 article [*No Silver Bullet: Essence and Accidents of Software Engineering*](https://www.infoq.com/articles/why-dsl-collection-anecdotes/#2) Turing Award winner Fred Brooks argues that

"...there is no single development, in either technology or management technique, which by itself promises even one order of magnitude [tenfold] improvement within a decade in productivity, in reliability, in simplicity."

He argues that modern, high-level programming languages have reduced the accidental complexity of a program so much, that more or less only the essential complexity is left. And essential complexity cannot be reduced. I think there are reasons to be skeptical about this statement. For one, there is still lots of accidental complexity in today's software systems. A key reason is that we don't just use one programming language, we use several of them in any given project, plus lots of frameworks and platforms. And second, the experience does show that a language that is high-level *relative to a domain*, aka a DSL, can lead to similar productivity improvements *in that domain* as a high-level general purpose language can bring over assembly language for programming in general.

Why is this? It is a consequence of the power of first-class concepts. When you make something a language concept instead of something expressed via composition of lower level concepts (aka programming), you can treat it specifically: a specific notation, specific analysis and error messages, specific IDE support, and specific translation. Of course this restricts flexibility. But you usually neither need nor want this flexibility in a DSL (except in places where you explicitly allow it). This is what makes it domain-*specific*.

None of this should come as a surprise! Computer science has always been about identifying the abstractions relevant for a class of problems and then finding ways of expressing these abstractions as clearly - as free from accidental complexity - as possible. Over the decades, our community has produced a vast range of languages that are specific to particular *technical* domains - from database queries over UI styling to control algorithms - and there's no reason why this same approach should not work for a wider range of (smaller and narrower) domains. Our experience, as showcased in this article, is testament to this.

*The purpose of abstraction is not to be vague, but to create a new semantic level in which one can be absolutely precise.* - Edsger Dijkstra

**Acknowledgements**

I want to thank my customers for allowing me to use company names. In particular, Wolfang Hager, Michael Kircher, Laurent Wiart, Etienne Vial and Andreas Wortmann. Dan Ratiu and Klaus Birken provided feedback on "their" tools, thank you! Thanks also to Nora Ludewig for editorial feedback on the paper. I also want to thank the reviewers of the ISOLA conferences for their feedback. Finally, I want to thank the team at InfoQ, in particular, Dylan Schiemann, Roxana Bacila and Ana Ciobotaru for their review and editing of this quite lengthy article.

**References**

1. K. Birken, D. Hunig, T. Rustemeyer, and R. Wittmann. Resource analysis of automotive/infotainment systems based on domain-specic models{a real-world example. In International Symposium On Leveraging Applications of Formal Methods, Verication and Validation, pages 424{433. Springer, 2010.

2. F. P. Brooks. No silver bullet: Essence and accidents of software engineering. IEEE Computer, 20:10{19, 1987.

3. M. Mernik, J. Heering, and A. M. Sloane. When and how to develop domain-specic languages. ACM computing surveys (CSUR), 37(4):316{344, 2005.

4. D. Ratiu, M. Gario, and H. Schoenhaar. Fasten: An open extensible framework to experiment with formal specication approaches. In 2019 IEEE/ACM 7th International Conference on Formal Methods in Software Engineering (FormaliSE), pages 41{50. IEEE, 2019.

5. A. Van Deursen, P. Klint, and J. Visser. Domain-specic languages: An annotated bibliography. ACM Sigplan Notices, 35(6):26{36, 2000.

6. M. Voelter. Language and ide development, modularization and composition with MPS. In GTTSE 2011, LNCS. Springer, 2011.

7. M. Voelter. Generic tools, specic languages. Self-Published, 2014. Dissertation.

8. M. Voelter. Fusing modeling and programming into language-oriented programming. In International Symposium on Leveraging Applications of Formal Methods, pages 309{339. Springer, 2018.

9. M. Voelter, A. v. Deursen, B. Kolb, and S. Eberle. Using c language extensions for developing embedded software: a case study. In Proceedings of the 2015 ACMSIGPLAN International Conference on Object-Oriented Programming, Systems, Languages, and Applications, pages 655{674, 2015.

10. M. Voelter, B. Kolb, K. Birken, F. Tomassetti, P. Al, L. Wiart, A. Wortmann, and A. Nordmann. Using language workbenches and domain-specic languages for safety-critical software development. Software & Systems Modeling, pages 1{24, 2018.

11. M. Voelter, B. Kolb, T. Szabo, D. Ratiu, and A. van Deursen. Lessons learned from developing mbeddr: a case study in language engineering with mps. Software & Systems Modeling, 18(1):585{630, 2019.

12. M. Voelter, D. Ratiu, B. Kolb, and B. Schaetz. mbeddr: Instantiating a language workbench in the embedded software domain. Automated Software Engineering, 20(3):339{390, 2013.

13. M. Volter. The design, evolution, and use of kernelf. In International Conference on Theory and Practice of Model Transformations, pages 3{55. Springer, 2018.